A Glance at Polymorphic Types

Lauri Alanko
`la@iki.fi`

Principles of Functional Programming
Advisor: Matti Nykänen

University of Helsinki
Department of Computer Science

February 25, 2000

Abstract

We examine cursorily the concept of *polymorphism* in programming languages. Polymorphism means the ability of a language construct, typically a function, to assume many types. We briefly survey different kinds of polymorphism, but concentrate mainly on *predicative parametric polymorphism*, which can be added as a relatively straightforward and consistent extension to simply typed lambda calculus.

Issues with type checking polymorphism are also considered, and a sample implementation is provided.

Contents

1 Introduction

Reusability and flexibility are some of the most important properties of good software. So are reliability and maintainability. One of the best ways to achieve the latter is to use a strong type system that uncovers many conceptual problems in a program as static type errors at compile time, as well as clearly documents the roles that variables and objects have in a program. However, it may appear that the use of strong static typing severely limits the reusability and flexibility of a program, since the types are hard-coded in. Polymorphism is a way to regain some, if not most, of this flexibility while still retaining the benefits of the type system.

We will have a quick look at different aspects of polymorphism. In section 2 we have an overview of the different forms in which polymorfism may manifest. In sections 3, 4 and 5 we examine parametric polymorfism from a slightly theoretical point of view. Finally, in section 6 we have a quick look at typechecking issues, accompanied by the sample implementation in appendix A.

2 Polymorphic concepts

Polymorphism means, literally, something of many forms. In the context of programming languages, it means some language construct that may be considered to have multiple types. These language constructs may be functions, operators or plain values, but henceforth we will simply speak of "polymorphic functions" since all of these can be considered functions[1].

2.1 Kinds of polymorphism

We can make a fundamental semantic distinction between two different kinds of polymorphism: A function can do essentially the "same thing" for all types, and the polymorphism merely makes the single function applicable to more than just one type. This is called *universal* polymorphism, since the same function can be used equally well for *all* types[2]. A contrast to this is *ad hoc* polymorphism. In ad hoc polymorphism, a function can only be used for a limited set of types, but its operation may vary according to the type. The name does not perhaps have very flattering overtones, and it is not even very accurate, but it has stuck in the terminology.

It is possible to further classify the subcategories of universal polymorphism:

Parametric polymorphism is the main topic of this paper. A parametrically polymorphic function takes (explicit or implicit) type parameters, the application of which results in a monomorphic (ie. not polymorphic) "instance" of the function.

[1]Specifically, an operator is typically a unary or binary function, while a simple value can be considered to be the result of the application of a nullary function.

[2]Unless we are dealing with bounded type parameters, which are not discussed here.

Subsumption refers to the most important feature of *subtyping*. This means that subtypes of the formal parameter type of a function can also be used as the types of the parameters. Calling this seemingly self-evident mechanism "polymorphism" may sound like splitting hairs, but it does fit into the definition. Indeed, one cannot really have subtyping without polymorphism of some sort.

Likewise, there are different kinds of ad hoc polymorphism, as well:

Overloading means dispatching the function call to the proper "implementation", depending on the type of the parameter. Depending on the details of the type system and other practical concerns, this dispatching may be done at compile time or at runtime, if such a distinction is sensible.

Coercing means implicitly transforming an "incompatible" value that is being passed as a parameter into a value of a proper type. The actual way in which this transformation can vary, but usually the intention is that some kind of a higher-level semantic "meaning" is retained while the low-level "representation" is changed. A classic example is implicitly converting integers into real numbers for functions that expect reals as parameters.

2.2 Practical rationale

Before going to formalisms, it is worthwhile to stop to think the reasons that make polymorphism an attractive feature in a practical language.

One of the most fundamental concepts in *any* kind of computer programming is abstraction. In a, well, abstract sense, abstraction means separating the operation from the operand. A simple form of abstraction is provided by functions, which allow a computation to be generalized and used on all kinds of values.

In a typeless (or sufficiently leniently typed) language, the computation can be performed on all kinds of sensible values[3]. However, in a strongly typed language, we are forced to lose some of this abstraction: a function has to have a type, which means that it cannot be used on *all* kinds of values. In a simple type system there is no way for a function to be generalized to work on many kinds of types, even if it did make sense.

Polymorphism is a mechanism intended to remedy this. The central idea of polymorphism is to rigorously specify, what things types have in common, and allow functions to make use of this commonality. For example, any values that can be hashed and compared for equality can be used as keys in a hash table. Thus polymorphism allows us to write code for manipulating hash tables only once, and allow the same code to be used for many kinds of key types. Because polymorphism always follows strict rules, strong typing is not compromised[4].

[3] And often on senseless values, too.

[4] Or at least it need not be. There are many weakly typed languages, too, with some kind of polymorphism.

If we view functions as abstractions over values, we can view polymorphism as abstractions over types. This is the topic of the next section.

3 Parametric polymorphism

3.1 Type functions

Parametric polymorphism, essentially, means that we extend a simple type system by having a new kind of construct: a function from types to values. We can represent this by means of ordinary lambda abstraction, with the only difference being that the parameter is a type, not a term. We write the type of a type parameter as T. A function of this kind can be applied to a type and it can be applied using ordinary β-reduction. It should be noted that the introduction of type parameters means that we have to extend our concept of "typing context" to include type variables as well, and also allow term variables to have a type bound to a type variable in their context. The formal rules for typing contexts will be presented shortly.

For example, let us define a simple polymorphic function, **twice**:

$$\text{twice} \equiv \lambda t : T.\lambda f : t \to t.\lambda x : t.f(f\ x)$$

This defines a polymorphic function that takes a single type parameter and returns a "self-compose" -function for functions from that type to that type. We can have a function on natural numbers:

$$\text{inc} \equiv \lambda y : Nat.(y + 1)$$

Now, we can apply this function to a Nat-typed self-composing function, and apply the resulting function to some number:

$$\begin{aligned}
&\text{twice}\ Nat\ \text{inc}\ 1 \\
\equiv\ & (\lambda t : T.\lambda f : t \to t.\lambda x : t.f(f\ x))\ Nat\ \lambda y : Nat.(y+1)\ 1 \\
\to^{\beta}\ & (\lambda f : Nat \to Nat.\lambda x : Nat.f(f\ x))\ \lambda y : Nat.(y+1)\ 1 \\
\to^{\beta}\ & (\lambda x : Nat.(\lambda y : Nat.(y+1))((\lambda y : Nat.(y+1))\ x))\ 1 \\
\to^{\beta}\ & ((\lambda y : Nat.(y+1))((\lambda y : Nat.(y+1))\ 1) \\
\to^{\beta}\ & (((\lambda y : Nat.(y+1))\ 1) + 1 \\
\to^{\beta}\ & 1 + 1 + 1
\end{aligned}$$

So there is really nothing magic in the use of types as parameters.

What is the type of **twice**? We cannot use the $\to$ -notation, because the return type depends on the parameter. Thus, the actual type of **twice** is a function from a type to a type[5]. Due to the way we can interpret functions as relations, and relations as sets of

[5]Note that we really mean that the type of **twice** is a *function*, not a function type.

pairs, a general product, Π, is appropriate for representing the type(s) of a polymorphic function. Thus, we can write the type of **twice** as:

$$\text{twice} : \Pi t : T.(t \to t) \to t \to t$$

This is quite analogous to a λ-abstraction, except that it returns a type, not a value.

3.2 Implicit polymorphism

In the previous example, a polymorphic function had to be applied an explicit type parameter before real value parameters could be passed to it. However, it would seem possible to deduce the needed type parameter from the type of the value parameters. Indeed, this is often, albeit not always, possible.

The difference between explicit and implicit polymorphism is mostly a practical one. Having the type parameters added automatically saves typing. The underlying semantics are however closely related to those of explicit typing.

With implicit polymorphism, it makes sense to think that since a polymorphic type can be used without further ado in all places where a monomorphic instantiation of it can be used, it is in fact a subtype of all those instantiations. Thus it is actually sensible to consider a polymorphic type to indeed have all the instance type at the same time, and it is sensible to write the type with the universal quantifer $\forall$ instead of Π.

Moreover, though we won't touch the subject further, it is worth noting that an existentially qualified type would be the supertype of all its typed instantiations. Thus we get the following graph:

$$\begin{array}{ccc} & \nearrow \exists t : T.(t \to t) \to t \to t \nwarrow & \\ (Nat \to Nat) \to Nat \to Nat & & (Bool \to Bool) \to Bool \to Bool \\ & \nwarrow \forall t : T.(t \to t) \to t \to t \nearrow & \end{array}$$

This perhaps best demonstrates the typing relations between polymorphic types.

4 Predicative polymorphic calculus

We now examine in further detail predicative polymorphic lambda calculus, $\lambda^{\to,\Pi}$, to see that simple polymorphism does not make a type system horribly more complicated. The key property of predicative polymorphism is that types are clearly divided into two sets, or 'universes'. There is U_1, the universe of "simple types", ie. the types appearing in $\lambda^{\to}$. Then there is U_2, which is the universe of polymorphic types that use the Π construct. The use of U_2 is restricted in predicative polymorphism. This simplifies the semantics of $\lambda^{\to,\Pi}$, but also limits its usefulness.

It should be noted that although we only discuss constructs to form parameterized polymorphic functions, the predicative typing system is quite sound for ad hoc polymorphism as well.

4.1 Context rules

We start with the basic axioms and rules of well-formed type contexts. We use the notation $\Gamma\diamond$ to indicate that the context Γ is well-formed in $\lambda^{\to,\Pi}$, and $\Gamma \rhd P$ to indicate that Γ is well-formed and that the typing assertion P holds within it. First, the general rules for manipulating typing contexts:

$$\emptyset\diamond$$

This means simply that the empty context is always well-formed.

$$\frac{\Gamma\diamond}{\Gamma, t : U_1\diamond}\,(t \notin \Gamma)$$

That is, one can add to any context a new type variable that belongs to the simply typed universe.

$$\frac{\Gamma \rhd \sigma : U_i}{\Gamma, x : \sigma\diamond}\,(x \notin \Gamma)$$

And we can thus add a new type term variable as long as it has a valid type in *some* type universe, that is, either U_1 or U_2.

$$\frac{\Gamma, x : A\diamond}{\Gamma, x : A \rhd x : A}$$

The rule for asserting the type of some variable is nearly trivial, but is included for completeness here.

$$\frac{\Gamma \rhd A : B\, \Gamma, x : C\diamond}{\Gamma, x : C \rhd A : B}$$

This rule is needed to make the variables of a typing context "inherited" everywhere the context is used, even if the variable is not the last one in the context list.

4.2 Type rules

Those were the rules for the generic well-formedness of typing contexts. Now the rules for actually obtaining valid types. Firstly, there is a set of primitive types, or "type constants", which we mark with b. These belong in the universe of simple types.

$$\emptyset \rhd b : U_1$$

Next, we have the relatively straightforward rule for function types:

$$\frac{\Gamma \rhd \tau : U_1\, \Gamma \rhd \tau' : U_1}{\Gamma \rhd \tau \to \tau' : U_1}$$

A very important thing to note here is that functions can only be constructed out of simple types. *It is impossible to build lambda abstractions for polymorphic types in predicative polymorphic calculus.* This is probably the most fundamental limitation of predicative polymorphism, and it has several practical consequences in both the use and implementation of a predicative language.

Now we get to the interesting bits, the polymorphic types of U_2. Firstly, all simple types are also U_2 types, for practical reasons.

$$\frac{\Gamma \triangleright \tau : U_1}{\Gamma \triangleright \tau : U_2}$$

And we can build "real" U_2 types by Π-abstraction over U_1-types:

$$\frac{\Gamma, t : U_1 \triangleright \sigma : U_2}{\Gamma \triangleright \Pi t : U_1.\sigma : U_2}$$

The important thing to note here is that the polymorphic type built with the Π -abstraction can take only U_1 -types as parameters, but can return another U_2 -type. This is necessary for us to be able to do "type currying" and create parametric types with more than one parameter[6].

4.3 Term rules

Finally, we consider the terms of $\lambda^{\rightarrow,\Pi}$. We use a somewhat simplified syntax, where a type τ and a type variable t are implicitly assumed to belong to U_1 in their contexts, and a type σ belongs to U_2. Again, we have the "built-in" constant expressions that have some built-in type:

$$\emptyset \triangleright c : \sigma$$

A λ-abstraction works in the usual fashion:

$$\frac{\Gamma, x : \tau \triangleright M : \tau'}{\Gamma \triangleright (\lambda x : \tau.M) : \tau \rightarrow \tau'}$$

Again, note that the abstraction works only over U_1-types, which is a significant limitation. Function application is similarly straightforward:

$$\frac{\Gamma \triangleright M : \tau \rightarrow \tau' \quad \Gamma \triangleright N : \tau}{\Gamma \triangleright M\,N : \tau'}$$

[6]This raises the question if it is possible for a Π-abstraction type to return itself, thus creating an "infinite type". The predicative calculus is so restricted that this is not feasible. However, in other systems we may want to distinguish the "arity" of a polymorphic type, and only allow it to return a type of an arity that is one less than its own. This can be accomplished by introducing the concept of *kinds* where monomorphic types have a kind of $*$, polymorphic types with one type parameter have a kind of $* \rightarrow *$, etc. In this terminology, the predicative calculus only allows formal parameters whose type's kind is $*$.

The new terms are the polymorphic abstractions:

$$\frac{\Gamma, t : U_1 \triangleright M : \sigma}{\Gamma \triangleright \lambda t.M : \Pi t.\sigma}$$

Note here that the type σ may depend on the bound variable t, which is really what parameterized types are all about. Application of a polymorphic function is fairly analogous to application of an ordinary function:

$$\frac{\Gamma \triangleright M : \Pi t.\sigma}{\Gamma \triangleright M\tau : [\tau/t]\sigma}$$

This last rule may look slightly odd. Since M is a term of a polymorphic type, we apply a type to it. Since the resulting type is the result of the Π-abstraction, we have to indicate it directly by the replacement operation.

4.4 Let-extension

Upon closer examination of $\lambda^{\to,\Pi}$, we note that it really doesn't provide any facilities for additional abstraction over $\lambda^{\to}$. The only way we can acquire a term of a U_2-type is from a constant, or by direct λ-abstraction over a type parameter, and the only thing we can do with such a term is apply it to a type. So one is forced to write the polymorphic term and its parameter together, which defeats the utility that a supposedly generic language construct is supposed to provide. Not being able to pass polymorphic types as parameters is analogous to not being able to pass functions as parameters in simply typed λ-calculus.

This is an inherent weakness of predicative polymorphism, and this is why impredicative polymorphism is clearly more powerful[7]. However, as a practical measure, we can add a new syntactic construct into $\lambda^{\to,\Pi}$.

A `let` expression is of the form

$$\mathrm{let}\, x : \sigma = N \,\mathrm{in}\, M$$

And it has the obvious typing rule:

$$\frac{\Gamma, x : \sigma \triangleright M : \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright (\mathrm{let}\, x : \sigma = N \,\mathrm{in}\, M) : \tau}$$

The let-syntax appears also as a simple extension to $\lambda^{\to}$, where it is defined purely syntactically as a combined λ-abstraction and function application:

$$\text{let } x : \tau = N \text{ in } M \equiv (\lambda x : \tau.M)N$$

[7]However, predicative polymorphism can arguably be often implemented more efficiently, especially in the case of ad-hoc polymorphism, where the dispatching to implementation by type can be done in compile time.

However, since we cannot have λ-abstraction over U_2-types (to which σ implicitly belongs in our notation), this is not possible in the extended calculus, $\lambda^{\rightarrow,\Pi,\mathrm{let}}$. Therefore, we must define let as a primitive expression, with very simple operational semantics:

$$\text{let } x = M \text{ in } N \rightarrow^{let} [M/x]N$$

So ultimately the let-syntax is merely a way to write a (possibly polymorphic) expression in one place, and refer to it by a variable inside the body of the let-expression. This can be considered simply syntactic sugar, but it is rather essential for practical purposes.

5 Extended calculi

Although predicative polymorphism is fairly easy to understand from a semantic point of view, its shortcomings are obvious. We will have a brief look at other systems.

5.1 Impredicative polymorphism

The primary difference of impredicative polymorphism, $\lambda^{\rightarrow,\forall}$ from $\lambda^{\rightarrow,\Pi}$ is that the types are no longer divided into separate "universes", but all types can be used in a uniform fashion without restrictions. This means that polymorphic values can be passed as parameters, and consequently read from variables. In practical terms, this means real run-time polymorphism, while in $\lambda^{\rightarrow,\Pi}$, no polymorphic function could be used without immediately applying a type parameter to it.

This extension provides further programming flexibility, but it complicates the semantic analyzation of the language. For example, non-parametric polymorphism doesn't fit very easily in the impredicative calculus. However, many if not most of the "real" programming languages of today are at least in some sense based on a restricted model of $\lambda^{\rightarrow,\forall}$.

5.2 Types as values

Perhaps the most expressive polymorphic type system is the "*type* : *type*" system. While $\lambda^{\rightarrow,\forall}$ removed the distinction between monomorphic and polymorphic types, *type* : *type* removes the distinction between types and values altogether. Thus a type is a value, whose type is also a value. This is reminiscent of the fully reflective and metacircular type systems such as those in CLOS and SmallTalk. Though allowing the manipulation of types in all ways similarly to values allows for very expressive constructs, it is in practice impossible to typecheck statically. Thus it is a type system that somewhat defeats the purpose of type systems. This doesn't mean that there wouldn't be practical use for dynamic type system of this sort, though.

6 Typechecking

Typechecking a polymorphic language is understandably more complex than checking a simply typed language. However, as long as lambda terms have type annotations in their parameter terms, the checking procedure is still relatively trivial. Implementing full-fledged type inference, however, is considerably more troublesome.

We will consider briefly the issues arising in basic typechecking when adding polymorphism to the language.

Lambda abstractions can be checked as usual, checking the type of the body in a type context extended by the formal parameter. However, one must check that the formal parameter does not have a polymorphic type, as that would break the predicativity of the language.

Function applications do not need any additional checking. The impossibility to pass a polymorphic type as a parameter already results from the fact that no function can take such a type as a parameter.

Type abstractions are fairly interesting. A type abstraction is checked in a fashion similar to lambdas, however, instead of introducing a new variable in the type context, we introduce a new type variable. This type variable can then be used as an ordinary type in the new context. The resulting type is a Π-construct, where the returned type can be composed of bound type variables.

Type applications are somewhat akin to evaluating a function application. Instead of extending a run-time "environment" with new variables, we extend the type context with a type variable that is bound to the actual type that was passed as a parameter. The Π-type is then resolved in this typing context.

The most significant change from simply typed lambda calculus is that whereas in $\lambda^{\rightarrow}$ all type annotations of formal parameters were absolute and unambiguously specified the type, in $\lambda^{\rightarrow,\Pi}$ a type annotation can be a type variable, which must always be looked up from the current typing context.

7 Conclusions

It is obvious that polymorphism, in some form, is an essential feature of any non-trivial programming language. Modern mainstream programming languages have type systems with quite sophisticated ad hoc polymorphism, often in the form of an object system. However, parametric polymorphism, the ability to create new types, has mostly been fully utilized only in the realm of functional languages, leaving the static type systems of other languages often anguishingly inadequate. Thankfully, the situation seems to be slowly improving (for example, see [BOSW98] and [CS98] for very promising treatments on extending the Java language).

References

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler: **Making the future safe for the past: Adding Genericity to the Java Programming Language**. OOPSLA '98 proceedings, pp. 183-200.

[CAR85] Luca Cardelli: **On Understanding Types, Data Abstraction, and Polymorphism**. Computing Surveys, 17(4):471-522.

[CAR87] Luca Cardelli: **Basic Polymorphic Type Checking**, Science of Computer Programming 8/2, April 1987.

[CS98] Robert Cartwright and Guy Steele: **Compatible genericity with run-time types for the Java Programming Language**. OOPSLA '98 proceedings, pp. 201-215.

[MIT96] John C. Mitchell: **Foundations for Programming Languages**. MIT Press, 1996.

[PIE??] Benjamin C. Pierce: **Type Systems for Programming Languages**. Unpublished draft, 2000.

A A type checking implementation for $\lambda^{\rightarrow,\Pi}$

Attached is an implementation in Haskell for a simple type checker for predicative polymorphic lambda calculus with builtin boolean and natural types. There is no real parser, so terms have to be construced using Haskell's primitive data constructors. The examples should clarify this. There is no actual evaluator included to get the values that terms reduce to, but this should be trivial to add.

The only real function of interest in the module is `checktype`, which takes a term as a parameter and returns its type, or an error message if the type checking failed.